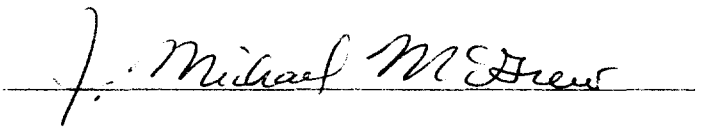
A Comparison of Two Computer Algorithms for Emulating a Vector Graphics Environment on a Dot Matrix Printer

An Honors Thesis (ID 499)

by

Christopher D. Bock

Thesis Director



Ball State University

Muncie, Indiana

May 1989

Table Of Contents

Introduction

This thesis will compare and contrast two algorithms that allow a graphics applications program to communicate with a dot matrix printer (a raster-based device) as a vector device.

Graphics devices are commonly grouped into two categories: vector devices and raster devices. A vector device is line-oriented -- e.g., a plotter. In order to plot a line on a vector device, the coordinates of the endpoints of the line are passed to the device and the line is plotted. However, a raster device can be thought of as being composed of a grid of points. A line plotted in a raster environment is an approximation of the line using these points. A dot matrix printer falls into this latter category of graphics devices.

The problem at hand is developing a library of graphics routines that allow a dot matrix printer to be addressed in a vector format. The complication that a dot matrix printer raises is that the printer prints a page "typewriter fashion", that is from the top of the page to the bottom, and, on each line, from left to right. This means that each line cannot be immediately printed like on a plotter -- each line must be stored until the page is ready to be printed at which time a line-by-line printing must occur.

The Two Algorithms

Two separate algorithms have been developed for this thesis. The first algorithm is what I refer to as the "conventional" method of solving the problem -- a grid is created in memory that equals the size of the printed page. When each line is generated, the line is plotted in the grid in memory. When the page is to be printed, the grid is simply copied onto the printer. The setback of this method is that it requires memory storage sufficient to hold an entire page of graphics data. It is very common on a small personal computer that not enough memory will exist to perform such an algorithm.

The second algorithm was an intuitive idea of my own in an effort to alleviate the memory requirements of the first algorithm. In this second method, a list of the endpoints is the only data that is kept, greatly conserving memory. When the page is printed line-by-line, the endpoint list is scanned for each print line and any lines that are within the current print line are plotted in that line only. The possible drawback to this method is that additional processing is done since the endpoint list is scanned for every print line.

Design Goals of the Graphics Library

The following were goals that were kept in mind while developing the graphics libraries:

1. a minimum amount of work should be required for the graphics application programmer.
2. the two versions of the library should be completely interchangeable: the graphics application programmer should not need to know which version of the library is being worked with.
3. "Standard" Epson printer codes should be used so that maximum compatibility is achieved.

The Dot Matrix Printer Environment

The dot matrix printer convention that will be used is the Epson graphics codes. The choice of Epson was made because the convention is probably the most popular, and many other brands of printers follow the Epson standard. The algorithms will be tested on two specific models: the MX-100, the first wide-carriage Epson produced; and an EX-800, a recent narrow-carriage model.

These printers are 9-pin printers, which means that the print head consists of a vertical column of nine pins. The print head moves laterally across the page, and when the pins are fired in the correct order, text or graphics output is produced.

In graphics mode, only eight of the nine pins are used for graphics. The ninth pin is only used for printing descenders on the text characters 'g', 'j', etc. This eight pin limitation was not randomly arrived upon: the normal unit of data communication is the byte, which contains eight bits each of which represents one of the pins (see figure 1). For instance, if the code 128 were sent to the printer in graphics mode, then the top dot would be printed. If 192 were sent, the top two dots would print, and so on. It would be awkward to try to address nine pins, so this eight pin limitation is reasonable. (Note: the newer Epson printers do have a special nonstandard mode that allows use of the ninth pin, yet each graphics column requires two bytes since a single byte is not enough to describe nine dots).

Printer Dot Density

The vertical density of dots on the Epson is fixed at 72 dots per inch because the pins are located 1/72 inch from each other on the print head assembly. Therefore, an 11 inch page would contain 792 dots vertically on the page. The horizontal density, which is determined by the rate the pins are fired as the print head moves across the page, is variable. In order to maintain compatibility with most Epson compatible printers, my routines support the original two graphics densities of 60 dots per inch (single density)

and 120 dpi (double density). On an 8 1/2 inch print line, that would mean lines with 510 and 1020 dots per line, respectively.

By multiplying the number of vertical dots per page by the number of dots across the page, we arrive at the size of the bitmap required (in dots). For an 8 1/2" x 11" page, a single density page requires a bitmap of 403,920 dots, while double density requires twice as many dots, 807,840. Since each dot is represented by a single bit, then the memory required would be the total number of dots divided by 8 (since 8 bits are in each byte). In the above example, memory required for the single density bitmap would be 50,490 bytes, and the double density bitmap, 100,980 bytes.

If we were working on wide computer paper (14 7/8" x 11"), then the bitmap sizes would be 706,860 dots for single density, 1,413,720 dots for double density; memory space required would be 88,358 bytes and 176,715 bytes, respectively.

As can be seen, the memory required for storing the bitmap can get quite large. On a small IBM/PC with only 256K of memory, chances are that the double density bitmap of an 14 7/8" x 11" page would probably not fit in memory, with the necessary overhead needed for MS-DOS and the graphics application program. It has not been but a couple of years since computers with 64K of memory were considered well-endowed. This low amount of memory would seriously

limit the size of the bitmap that could be created. It is this memory concern that sparked my intuition for the second version of my graphics library, which does not require a bitmap of the entire page to be stored in memory.

Epson Printer Graphics Codes

The following are the Epson printer codes that are used in the graphics library for printed output.

ESC @

Resets the printer to its power-on settings (type style, line spacing, etc.)

ESC A n

Sets the vertical line spacing to n/72", where n is sent as a single eight-bit value. Normally, for 6 lines per inch, the line spacing is 12/72"; similarly for 8 lines per inch, the line spacing is 9/72". Since we use 8 pins in graphics work, the line spacing is set to 8/72" so that after a carriage return, the uppermost print head pin will be immediately below the lowermost pin of the previous line.

ESC K n1 n2
ESC L n1 n2

These are the actual graphics commands -- K is used for single density (60 dots per inch) printing, and L for double density (120 dpi) printing. The two values n1 and n2 are eight bit values that when combined to form a sixteen-bit value (with n2 as the most significant byte) tell the printer how many bytes of graphic data will immediately follow the codes.

Number of Columns = (n2 * 256) + n1

Bresenham's Line Drawing Algorithm

Both versions incorporate Bresenham's line drawing algorithm, which is an algorithm that calculates the pixels required to represent a line on a raster device. An algorithm to do such a task can be easily derived that involves calculating a slope of the line and then computing pixels to plot. However when implemented, such an algorithm requires repetitive floating point operations, which slows the algorithm noticeably.

Bresenham derived an algorithm that requires only integer operations to approximate a line. In the algorithm, a loop steps through whichever range (x or y) is the smaller, setting one pixel per column. A simple integer operation is required on each column to determine which pixel to set on the other range. The code that is included in the two versions of the graphics library was adapted from the pseudocode of Bresenham's algorithm presented in the book "Graphics in C" by Nelson Johnson.

For a complete explanation of the Bresenham line drawing algorithm, see a graphics text such as Hearn/Baker.

Cohen-Sutherland Clipping Algorithm

When implementing a project such as this, the question arises as to what action needs to be taken if the line (-1,-1), (1,1) is to be plotted in the area with bounds (0,0), (5,5). One option would be to reject the line, on the

grounds that the coordinates are out-of-bounds. Another option is to "clip" the line, changing one or both of its endpoints such that the portion of the line that would have been within the region is retained. In the example above, the clipping would have changed the first endpoint to (0,0), thereby retaining the line as it would have existed within the region.

The Cohen-Sutherland clipping algorithm is based on encoding the endpoints with codes that describe their location relative to the active area. The line is then repetitively clipped to one of the boundaries of the region until the endpoints all lie either on or within the area. The equations used to "move" the endpoints are based on right triangles and preserve the slope of the line.

For a complete explanation of the Cohen-Sutherland clipping algorithm, see a graphics text such as Hearn/Baker.

Using the Graphics Library

The following library calls are available to the graphics application programmer:

Gopen(int dens, float width, float height)

"Opens" the graphics area for use. The first parameter is the horizontal density (constants LORES and HIRES are defined for this purpose); the second and third are float values that define the size of the output area in inches.

Gclear()

Clears the graphics area to an empty state. Gopen() automatically calls Gclear() upon opening the graphics area.

Gline(int x1, int y1, int x2, int y2, unsigned linestyle)

Plots a line from the pixel located at (x1,y1) to (x2,y2) using the linestyle specified in the last parameter. The linestyles G_SOLID, G_DASHED, and G_DOTTED are predefined for use. The bit pattern formed by the 16-bit unsigned integer is used as the line template.

Gprint(int prtr)

Prints the page on the printer specified by the parameter. Predefined constants PRN, LPT1, LPT2, and LPT3 are available for use -- PRN and LPT1 refer to the same printer.

Gclose()

Closes the graphics area and frees any memory previously allocated.

Version 1: The Bitmap Implementation

The first version of the graphics library is what I call the "conventional" implementation -- a bitmap is created for the entire graphics area and lines are plotted in the bitmap at the time Gline() is called. To print a copy of the area, Gprint() simply dumps the bitmap to the printer.

Gopen()

The main purpose of Gopen() is to allocate a block of memory of the necessary size to hold the requested graphics area. Gopen() is passed the printer density, and the width and height of the area desired. The following assignments are made:

```
density = dens;
pageheight = (int) height * LPI;
pagewidth = (int) width * density;
bitmapsize = (long) pageheight * pagewidth;
```

The first assignment is simply to place the density parameter, **dens**, into a global variable, **density**, for use in the other functions. Note that the global variables are declared static so that they are global only to the graphics library code -- they are not available outside of these routines.

The global variables **pageheight** and **pagewidth** are then calculated based on the input parameters of **height** and **width**, which are expressed in inches. **LPI** is a defined constant which expresses the number of printer lines per

vertical inch. Therefore, **pageheight** gets assigned the number of printer lines (each of which is eight dots high), and **pagewidth** gets the number of columns (or dots) across the page.

The global variable **bitmapsize** is simply the page height multiplied by the page width, and cast into the long integer (32 bit) data type -- this is because it is very possible for the size of a bitmap to be larger than an unsigned sixteen bit value would allow, which is 65,536 bytes.

Due to the segmented memory architecture in the 8088-family of microprocessors, normally a data block can be at most 65,536 (64K) large. However, Turbo C allows blocks to be allocated that are greater than this limit, and these blocks are then referenced through a far pointer. The farmalloc() call takes care of allocating memory for the bitmap -- an error is returned if there is not enough memory available for allocation.

Global variables **minx**, **maxx**, **miny**, and **maxy** are then calculated to give the pixel range available for plotting. A call to Gclear(), which zeroes the entire contents of the bitmap, finishes the Gopen() function.

Gclear()

Gclear() sets up a loop that goes through every byte in the bitmap for the graphic area and sets those bytes to

zero. A more compact (and possibly faster) method of doing this would be by use of the memset() library function, yet I questioned its portability, so I simply set up a for loop to accomplish the task.

Gclose()

The purpose of Gclose() is to terminate the graphics processing and return any allocated memory back to the operating system. The library procedure farfree() is used to free the memory block.

Gline()

The purpose of Gline() is to plot a line in the bitmap. The endpoints of the lines are passed to Gline() as is the requested linestyle.

After making sure a Gopen() has been done, the endpoints provided are checked to see if they fall in the bounds designated by (minx, miny), (maxx, maxy). If the line is not wholly contained in this region, then the line is clipped to the boundaries of the region using the Cohen-Sutherland clipping algorithm. If the line was never within the region, then Gline() simply returns with an error code of G_CLIP.

After a set of valid endpoints are obtained, then the individual pixels that comprise the line are computed using Bresenham's algorithm. The only twist in the Bresenham

algorithm is the introduction of a linestyle. When Gline() is called, an unsigned 16-bit value is passed that determines the style of the line to be plotted. The bit pattern formed by the 16-bit value is used as the linestyle of the line to be plotted. The basic algorithm for using the linestyle parameter is as follows:

```
while (more pixels to plot)
     generate the next pixel to plot
     if (rightmost bit of linestyle is set) then
          plot pixel
     else
          do not plot pixel
     endif
     perform a circular shift of linestyle to the right
end while
```

Since a circular shift is not implemented in C, the following code is integrated into the Bresenham algorithm to perform the function using the available arithmetic shift (which nondiscriminately shifts a zero in the appropriate end):

```
if (linestyle & 0x0001)
     linestyle = (linestyle>>1) | 0x8000;
else
     linestyle >>= 1;
```

The first line checks to see if the right-most bit is set (which means that a one bit must be shifted in from the left). If so, the linestyle is shifted right arithmetically by 1 bit and the left-most bit is ORed to 1; otherwise, since the right-most bit is 0, an arithmetic shift right of 1 bit will suffice.

As the coordinates of each pixel are calculated (which are expressed in pixel units), then the function Gsetpixel()

is called with these coordinates. Gsetpixel() first makes sure that a Gopen() has been performed, and then that the pixel coordinates passed to it are indeed valid. It should be noted that at this point, Gsetpixel() should never see invalid coordinates -- the line should have been validated and clipped if necessary to make the coordinates valid.

The following code is then executed to set the single pixel designated as (x,y):

```
ydiv8 = y >> 3;
offset = (ydiv8 * pagewidth) + x;
mask = 0x80 >> (y - (ydiv8<<3));
*(map+offset) |= mask;
```

The first calculation (ydiv8) calculates which printer line contains the pixel to be plotted. Since each printer line contains 8 pixels, an integer division by 8 of the y coordinate will return the line number of the printer line where the pixel will be plotted.

The offset into the bitmap is then calculated by taking the line number obtained from the previous step, multiplying it by the size in pixels of a horizontal print line (thereby obtaining the offset to the beginning of that particular line in the bitmap) and then by adding the x coordinate which locates the particular byte in the bitmap.

Next, the remainder of the integer division by 8 is needed to determine which bit of the eight bits possible is the correct bit to set. Since the top pin of the print head is represented by 128, the next by 64, etc., and the remainder will be in the range 0-7 (from top to bottom), all

that is needed is to shift 128 (represented by 0x80) right by the remainder amount.

This right shift results in a mask value that contains only one bit set, which corresponds to the pixel that needs to be set. All that is required now is to take the particular byte in the bitmap, which is addressed as *(map+offset) and logically OR the mask, thereby preserving any of the other bits that may have previously been set.

Gprint()

Printing out the bitmap is not really a difficult procedure; the bitmap just needs to be dumped a print line at a time. After experimenting with the program, I discovered that the Epson printer was not intelligent enough to know when it was printing a blank line. The print head blindly goes across the page printing nothing, so I added a procedure that would scan the print line prior to output so that if the line is blank, I could skip the output loop and go to the next line.

For simplicity I used BIOS calls to handle the printer output. I encapsulated them in their own separate procedures Gprinterstatus(), Gprinterinit(), and Gprintchar(). They all expect an integer parameter that tells which printer number to use for output. This value is passed to Gprint() from the application program, and is passed along to these BIOS routines.

The only problem in using the BIOS calls is the case where the printer may time-out due to being switched off-line, or running out of paper, etc. Gprint() checks the printer status before printing and will return the error code G_NOTRDY if a printer error is detected; however, once the printing starts, if a printer error occurs, then the printer will eventually time-out, giving an "Abort, Retry, Ignore" message common to MS-DOS.

The printer codes used were described in an earlier section of this paper. After checking the printer status and then initializing the printer, we send the code sequence to set the line spacing to 8/72 inch, which we need for graphics printing.

The following piece of code separates a sixteen-bit integer into its two eight-bit portions as required by the two graphics codes:

```
Gprintchar(pagewidth & 0x00FF,prtr);
Gprintchar(pagewidth >> 8,prtr);
```

The first statements mask out the top order 8 bits and sends the low-order byte, while the second statement shifts the top eight bits into the lower 8 bits. Since the first parameter of Gprintchar() is defined as **char** (an eight-bit value), the AND operation performed in the first statement would not be necessary -- Gprintchar() would take the low order byte of the integer and disregard the top byte, yet performing this AND seems to make it a little clearer what is happening.

Note that the entire output loop is enclosed in the IF statement:

```
if (!Gblankline(bitmap))
```

so that the output loop will only be executed if the blank is not blank. The code for Gblankline() simply sets up a loop for the current print line. As soon as a non-zero byte is found, the function returns with a FALSE value, meaning the line is not blank. Otherwise, if the loop falls through, then a TRUE condition is returning meaning the line indeed has only zero values in it.

After the printing is complete, a top-of-form code is sent to advance the paper to the next form, and then the printer reset code sequence is sent so that the printer is restored to power-up conditions. Basically this is to return the line spacing back to six lines per inch (12/72 inch) instead of the 8/72 inch we had set for the graphics printing.

Version 2: The "Raster" Implementation

I call the second implementation the "raster" version because it retains the data about the lines instead of immediately plotting the lines and then discarding the data about the lines. The lines are stored in a linked list and a bitmap for the entire area is not required. A bitmap for a single print line is required, yet at most only around 1,500 bytes would be required for the widest print line. Therefore, memory size limits the number of lines that can be plotted, not the size of the printed area.

The theory of this algorithm is as follows. Each time Gline() is called from the application program, the endpoints and linestyle are added to a dynamically allocated linked list. Before being added to the list, coordinate validity is checked, and clipping is performed if necessary.

When Gprint() is called, the page is printed one line at a time. The following algorithm describes the printing process:

```
while (more lines to print)
     clear print line
     for each line in the linked list
          if line intersects current print line
               plot part of line in current print line
          end if
     end for
     output print line
end while
```

As can be seen, the processing is delayed until the printing stage, which requires that the list of lists be scanned for each print line before printing. The possible downfall to

this algorithm is that it might be slower because of the additional processing that is performed.

Gopen()

The code for Gopen() is basically the same as for the bitmap version with the exception that only a single print line is allocated as opposed to the complete print area. Since the size of a single print line will most definitely be less than 65,536 bytes, malloc() is used, which returns a near pointer to the allocated memory.

Gclear()

In order to clear all of the line data, Gclear() traverses the linked list and releases each node back to the operating system.

Gclose()

Gclose() is exactly the same as the bitmap version, except that free() is used instead of farfree() since the buffer was allocated using malloc() and near pointers.

Gline()

The complexity of the bitmap version's Gline() was basically moved to the Gprint(), so that the Gline() is basically a simple linked list insertion function. The endpoints are first checked for bounds and clipped if

necessary. Memory is then allocated for a new node and the data is inserted into the node, the node is linked into the existing list, and the function returns a normal completion status.

Gprint()

The algorithm for Gprint() was listed above. It is clear that all of the processing has been placed in Gprint(), such as Bresenham's algorithm and a specialized clipping routine that determines where a particular line intersects the current print line. Again, right triangle equations are used to clip the endpoints of the line to temporarily map into the current print line. See Figure 1 in the appendix for a diagram of the equations involved.

An interesting bug occurred when I was debugging this version. On the printouts, non-vertical lines were not solid, but were missing dots every eighth row. The dots that were missing were all located on the same rows on the page, so obviously a pattern was present. Here is the explanation of the problem:

When clipping a line into the current print line, an x value was calculated for the y value represented by the top pin of the print head (y+0), and an x value for the bottom pin of the print head (y+7). A line was then drawn between the two. On a sloping line, a gap occurred because values of y were not being considered between the bottom pin of the

print head and the top pin of the next print line. To solve this problem, instead of calculating an x value for the bottom pin of the print head (y+7), the line was carried on to another pin (one that really doesn't exist, y+8). This caused the gaps to be filled in; however, invalid coordinates (like a dot on the ninth pin of the print head) were generated. Instead of programming around this special case, the return value of Gsetpixel(), which would be where the invalid coordinate is caught, is simply ignored.

The code from Bresenham's algorithm is unchanged from the bitmap version, because only the Gsetpixel() function needed to be rewritten since the pixel to be plotted would exist only within a single print line. As in the bitmap version, the bounds of the coordinates are first checked by Gsetpixel(), and then the pixel is set using the less complicated:

```
*(linemap+x) |= (0x80 >> y)
```

which sets bit y of the xth byte in the print line.

A Comparison of the Two Algorithms

There are two areas in which I will compare the two algorithms: memory requirements and speed. These two comparison points have traditionally been inversely related: an increase in speed usually meant a decrease in memory efficiency; likewise, an increase in memory efficiency usually meant a deterioration in execution time. As noted in a previous section, a decrease in execution time was expected in the raster algorithm, due to the additional processing required.

Memory Requirements

The following table compares the memory requirements of the two algorithms (all sizes are in bytes). Note that the bitmap requires the same amount of memory irregardless of the number of lines to be plotted; however, the memory requirements for the raster version vary with the number of lines:

	8" x 10" Image Area		13" x 10" Image Area	
	LORES	HIRES	LORES	HIRES
Bitmap:	43,200	86,400	70,200	140,400
Raster:				
100 Lines	1,880	2,360	2,240	3,040
1,000 Lines	14,480	14,960	14,780	15,560
10,000 Lines	140,480	140,960	140,780	141,560

The following functions describe the memory requirements for the two algorithms:

Bitmap Version = (page length, inches)(9 lines/inch) *
(page width, inches)(horiz. density)

Raster Version = (page width, inches)(horiz. density) +
(14 bytes)(number of lines)

...where horiz. density = 60 for low density
120 for high density

The following table lists the "break-even" point between the two algorithms, which is the point where the raster algorithm requires the same amount of storage as the bitmap version. A plot that contains less than the stated number of lines will require less memory with the raster version; a plot with more lines will require less memory with the bitmap version.

	Number of Lines	Memory (bytes)
8 x 10 inch area:		
Low density:	3,051	43,200
High density:	6,111	75,600
13 x 10 inch area:		
Low density:	4,959	70,200
High density:	9,917	140,400

Execution Time Comparison

The following table lists the execution times obtained when testing the algorithms using a varying number of lines. Each line was a diagonal from the top-left position to the bottom-right position of the page. Even though all lines were identical, the algorithms still churned through each line. The tests were performed on a 12Mhz AT-compatible and an Epson EX-800, which has a draft print speed of 300 characters per second. All execution times are in seconds.

	Bitmap Version	Raster Version
100 lines	42.75 sec	35.16 sec
1,000 lines	102.42	70.33
10,000 lines	699.78	185.93

As can be seen, the raster version outperformed the bitmap version in all cases. This was a very unexpected result. Originally, I felt the raster version would be at a disadvantage due to the amount of processing that had to be done for each print line. The timing of this processing leads to the explanation of the result.

In the bitmap version, all processing is performed first, and when a plot is requested, the bitmap is simply dumped to the printer. In the raster version, processing is performed between each line and this is where the speed advantage is gained.

When a printer prints data, the computer sends a line of data, waits for the line to be printed, and then proceeds to the next line. In the bitmap version, this period between lines is simply a busy wait -- the computer is merely waiting on the printer to print the previous line in order to accept the next line. However, in the raster version, the computer processes the data for the next line during this period, therefore making the computer productive during this waiting time.

Future Enhancements

The graphics library presented in this thesis is only a start of what could be expanded to be a complete graphics library. Following are my comments on some enhancements that would make this library a general-purpose graphics library.

Other Graphics Primitives

The inclusion of other graphics primitives other than the line would give the graphics applications programmer more flexibility. Most other objects, such as circles, arcs, polygons, can all be implemented such that these object are simply a collection of lines. A circle is simply an n-sided polygon, where as n increases, the roundness of the circle approximates a true circle. These additional library calls would simply make use of Gline() to construct more complex objects.

Area Fills

The ability to fill an area is often desired, both using a solid fill or maybe a patterned fill. This would not be difficult to implement on the bitmap version, as a scan line algorithm would work well; however, in the raster version a different sort of algorithm would need to be employed. It would be similar to the process involved in the my raster algorithm, in that it would work with filling

a polygon given only the edges. Several algorithms exist for both of these fills; consult a graphics text for their implementation.

Color

The Epson EX-800 that was used for this project has the ability to print up to 7 colors. A possible implementation of color using the bitmap method would be to enlarge the bitmap such that each pixel would be represented by 3 bits instead of a single bit as implemented now. Three bits would allow 8 combinations -- one for blank and the other seven would represent the colors. Some scheme would need to be devised to decide what color results when lines of different colors intersect.

The raster version would simply get another field added into the LINESTRUCT structure that designates color. A possible implementation would allow the print head to make up to seven passes on a single line, each pass printing a separate color. Using this method, a true color blending would take effect if lines of different colors intersected.

Other Printers

Of course for a graphics library to become widely-used, it must support a wide range of printers. Conversion to a different printer should not be difficult since all of the graphics codes are well-documented. A nice feature would allow the printer type to be specified when Gopen() is

called. That way, a common interface could be used for a variety of printers and even plotters. Plotters would be easily supported -- instead of worrying about computing the line via Bresenham's algorithm, the library could just generate the appropriate plotter command to plot the line.

Conclusion

This thesis project has been an enjoyable as well as a worthwhile experience. Computer graphics is a fun topic, one in which the work involved is hidden by the enjoyment obtained from the results.

I would like to thank Dr. McGrew, who not only sparked my interest in computer graphics through his graphics course, but who also introduced me to the C programming language in an earlier course. His help has been priceless and much appreciated.

Appendix

The following items are presented in this appendix:

List of Reference Texts

Figure 1. Computation of Intersection Points for the Raster Algorithm

Header File for the Graphics Library

Source Code for the Bitmap Version of the Graphics Library

Source Code for the Raster Version of the Graphics Library

Examples of Computer Graphics Generated by These Two Algorithms

Reference Texts:

Hearn, Donald, and Baker, M. Pauline (1986). Computer Graphics. New Jersey: Prentice-Hall.

Johnson, Nelson (1987). Advanced Graphics in C: Programming and Techniques. California: Osborne/McGraw Hill.

```
/*************************************************************/
/*                                                           */
/*       DOT MATRIX PRINTER GRAPHICS LIBRARY                 */
/*                                                           */
/*                                                           */
/*       Author: Christopher Bock                            */
/*                                                           */
/*       Course: Honors College Thesis, ID 499               */
/*                                                           */
/*               Ball State University                       */
/*                                                           */
/*               Spring Semester 1989                        */
/*                                                           */
/*************************************************************/

/*************************************************************/
/*               HEADER FILE                                 */
/*************************************************************/

/*****************************************/
/*       ERROR CODES                     */
/*****************************************/
#define G_OK              0
#define G_NOINIT          1
#define G_NOMEM           2
#define G_NOTRDY          3
#define G_BOUNDS          4
#define G_CLIP            5

/*****************************************/
/*       CONSTANTS                       */
/*****************************************/

/* for printer densities */
#define LORES             60
#define HIRES             120

/* define labels for printers */
#define PRN               0
#define LPT1              0
#define LPT2              1
#define LPT3              2

/* line styles */
#define G_SOLID           0xFFFF
#define G_DASHED          0xF0F0
#define G_DOTTED          0xCCCC

/*****************************************/
/*       FUNCTION PROTOTYPES             */
/*****************************************/
int Gopen(int,float,float);
int Gclear();
int Gline(int,int,int,int,unsigned);
int Gprint(int);
int Gclose();
```

```
/*************************************************************/
/*                                                           */
/*       DOT MATRIX PRINTER GRAPHICS LIBRARY                 */
/*                                                           */
/*                                                           */
/*       Author: Christopher Bock                            */
/*                                                           */
/*       Course: Honors College Thesis, ID 499               */
/*                                                           */
/*               Ball State University                       */
/*                                                           */
/*               Spring Semester 1989                        */
/*                                                           */
/*************************************************************/

/*************************************************************/
/*               BIT MAP VERSION                             */
/*************************************************************/

/* needed for farmalloc(), farfree() */
#include <alloc.h>
/* needed for biosprint() */
#include <bios.h>

#define max(x,y)        ((x)>(y)?(x):(y))
#define abs(x)          ((x)>0?(x):-(x))

/*****************************************/
/*      ERROR CODES                      */
/*****************************************/
#define G_OK            0
#define G_NOINIT        1
#define G_NOMEM         2
#define G_NOTRDY        3
#define G_BOUNDS        4
#define G_CLIP          5

/*****************************************/
/*      CONSTANTS                        */
/*****************************************/

#define TRUE            1
#define FALSE           0
#define ESC             27
#define HUGENULL                ((char huge *) 0)

/* constants for line densities */
#define LORES           60
#define HIRES           120

/* various printer constants */
#define LPI             9
#define DPL             8
#define VERTDPI         72
```

```
/* define labels for printer ports */
#define PRN             0
#define LPT1            0
#define LPT2            1
#define LPT3            2

/* line styles */
#define G_SOLID         0xFFFF
#define G_DASHED        0xF0F0
#define G_DOTTED        0xCCCC

/*****************************************/
/*      GLOBAL VARIABLES                 */
/*                                       */
/* all global variables are static so    */
/* they are unknown/unaccessible         */
/* outside of this file.                 */
/*****************************************/

static int density;                     /* printer density */
static unsigned long bitmapsize;        /* size of page bitmap */
static char huge * map = HUGENULL;      /* pointer to bitmap */
static int minx, maxx;                  /* minimum/maximum for x */
static int miny, maxy;                  /* minimum/maximum for y */
static int pageheight;                  /* height of bitmap in lines */
static int pagewidth;                   /* width of bitmap in pixels */


/*****************************************/
/* Gopen()                               */
/*      sets up a bitmap of the size     */
/*      needed for the 'density'         */
/*      provided.                        */
/*                                       */
/* Return Values:                        */
/*   G_OK       success                  */
/*   G_NOMEM    not enough memory        */
/*****************************************/
int Gopen(dens,width,height)
int dens;
float width,height;
{
        /* save density in global variable */
        density = dens;

        /* calculate page height & width in suitable units */
        pageheight = (int) height * LPI;        /* in lines */
        pagewidth = (int) width * density;      /* in pixels */

        /* calculate size of bitmap needed */
        bitmapsize = (long) pageheight * pagewidth;

        /* allocate the memory as a far block */
```

```
        if ((map = farmalloc(bitmapsize)) == HUGENULL)
           return (G_NOMEM);

        /* calculate minimums & maximums for x,y */
        minx = miny = 0;
        maxx = pagewidth-1;
        maxy = (pageheight * DPL)-1;

        /* clear map */
        Gclear();

        return (G_OK);
}

/*****************************************/
/* Gclose()                              */
/*       releases the memory allocated   */
/*       to the bitmap.                  */
/*                                       */
/* Return Values:                        */
/*   G_OK        success                 */
/*   G_NOINIT    not initialized w/ open */
/*****************************************/
int Gclose()
{
        /* check to see if a Gopen has been done */
        if (map == HUGENULL)
           return (G_NOINIT);

        /* release memory from bitmap */
        farfree(map);

        /* mark map as pointing to NULL */
        map = HUGENULL;

        return (G_OK);
}


/*****************************************/
/* Gclear()                              */
/*       clears the bitmap to an 'empty' */
/*       condition. (all locations = 0)  */
/*                                       */
/* Return Values:                        */
/*   G_OK        success                 */
/*   G_NOINIT    not initialized w/ open */
/*****************************************/
int Gclear()
{
        long i;                 /* index variable */
        char huge * mapprtr;    /* pointer to bitmap */

        /* check to see if a Gopen has been done */
```

```
        if (map == HUGENULL)
           return (G_NOINIT);

        /* set mapprtr equal to map */
        mapprtr = map;

        /* loop thru bitmap setting all locations to 0 */
        for (i=bitmapsize; i>0; i--)
           *mapprtr++ = '\0';

        return (G_OK);
}


/*****************************************/
/* Gprint()                              */
/*      prints the page to the printer.  */
/*                                       */
/* Return Values:                        */
/*   G_OK        success                 */
/*   G_NOINIT    not initialized w/ open */
/*   G_NOTRDY    prtr not ready          */
/*   G_BOUNDS    prtr num out of bounds  */
/*****************************************/
Gprint(prtr)
int prtr;       /* printer number (0=LPT1,1=LPT2,2=LPT3) */
{
        char huge * bitmap;     /* pointer to bitmap */
        int lines, columns;     /* index variables */
        char prcode;            /* control code for graphics */

        /* check to see if a Gopen has been done */
        if (map == HUGENULL)
           return (G_NOINIT);

        /* check to see if printer number is valid */
        if ((prtr < 0) || (prtr > 2))
           return (G_BOUNDS);

        /* initialize bitmap to point to real bitmap */
        bitmap = map;

        /* see if printer is ready */
        if (Gprinterstatus(prtr))
           return (G_NOTRDY);

        /* initialize the printer */
        Gprinterinit(prtr);

        /* set the printer up for correct vertical spacing */
        Gprintchar(ESC,prtr); Gprintchar('A',prtr); Gprintchar(8,prtr);

        /* go through each line of the page */
        for (lines=0; lines<pageheight; lines++) {
```

```
            /* check for a blank line */
            if (!Gblankline(bitmap)) {

               /* send the graphics initialization string */
               switch (density) {
                  case LORES: prcode = 'K'; break;
                  case HIRES: prcode = 'L'; break;
                  }

               Gprintchar(ESC,prtr); Gprintchar(prcode,prtr);

               /* send low order, then high order byte */
               Gprintchar(pagewidth & 0x00FF,prtr);
               Gprintchar(pagewidth >> 8,prtr);

               /* do the entire row */
               for (columns=0; columns<pagewidth; columns++)
                  Gprintchar(*bitmap++,prtr);
               }
            else
               bitmap += pagewidth;

            /* advance to the next line */
            Gprintchar(13,prtr); Gprintchar(10,prtr);
            }

            Gprintchar(12,prtr); /* go to TOF */
            Gprintchar(ESC,prtr);        /* send software reset to printer */
            Gprintchar('@',prtr);

}

/*****************************************/
/* Gblankline()                          */
/*      checks to see if the current     */
/*      line is blank (contains all      */
/*      zero bytes).                     */
/*                                       */
/* Return Values:                        */
/*   TRUE       line is blank            */
/*   FALSE      line is not blank        */
/*****************************************/
static int Gblankline(bufprtr)
char huge * bufprtr;
{
        int i;          /* index variable */

        i = pagewidth;
        while (i--)
           if (*bufprtr++)
              return (FALSE);

        return (TRUE);
```

```
}

/*****************************************/
/* Gprinterstatus()                      */
/*      returns status of the printer.   */
/*                                       */
/* Return Values:                        */
/*   0          printer ready            */
/*   non-0      problem at prtr          */
/*****************************************/
static int Gprinterstatus(prtnum)
int prtnum;
{
        return(biosprint(2,0,prtnum) & 0x29);
}


/*****************************************/
/* Gprinterinit()                        */
/*      Uses BIOS to initialize printer  */
/*                                       */
/* Return Values:                        */
/*   G_OK       success                  */
/*****************************************/
static int Gprinterinit(prtnum)
int prtnum;
{
        biosprint(1,0,prtnum);
        return (G_OK);
}


/*****************************************/
/* Gprintchar()                          */
/*      prints a byte to the printer     */
/*                                       */
/* Return Values:                        */
/*   G_OK       success                  */
/*   G_NOTRDY   prtr not ready           */
/*   G_ESC      user pressed ESC         */
/*****************************************/
static int Gprintchar(ch,prtnum)
char ch;
int prtnum;
{
        biosprint(0,ch,prtnum);
        return (G_OK);
}


/*****************************************/
/* Gsetpixel()                           */
/*      sets a single pixel.             */
/*                                       */
```

```
/* Return Values:                          */
/*   G_OK        success                   */
/*   G_NOINIT    not initialized w/ open   */
/*   G_BOUNDS    x or y out of bounds      */
/******************************************/
static int Gsetpixel(x,y)
int x,y;
{
        int ydiv8;              /* will hold int(y/8) */
        long offset;            /* offset into bitmap */
        unsigned char mask;     /* mask value for setting pixel */

        /* check to see if a Gopen has been done */
        if (map == HUGENULL)
           return (G_NOINIT);

        /* check x,y for bounds */
        if ((x<minx) || (y<miny) || (x>maxx) || (y>maxy))
           return (G_BOUNDS);

        /* shift y right by 3, effectively divided by 8 */
        ydiv8 = y >> 3;

        /* calculate offset into bitmap */
        offset = (((long) ydiv8) * ((long)pagewidth)) + ((long) x);

        /* compute 'mask' byte for setting pixel */
        mask = 0x80 >> (y - (ydiv8<<3));

        /* now set pixel */
        *(map+offset) |= mask;


        return (G_OK);
}

/******************************************/
/* Gencodepoint()                         */
/*      Part of Cohen-Sutherland          */
/*      Clipping algorithm.               */
/*                                        */
/* Return Values:                         */
/*   encoded value for endpoint           */
/******************************************/
static unsigned Gencodepoint(x,y)
int x,y;
{
        unsigned code = 0;      /* used for constructing code */

        if (x<minx) code |= 0x01;
        if (x>maxx) code |= 0x02;
        if (y<miny) code |= 0x04;
        if (y>maxy) code |= 0x08;
```

```
        return (code);
}

/*****************************************/
/* Gclip()                               */
/*      Cohen-Sutherland algorithm       */
/*      Clips endpoints of a line that   */
/*      falls outside of the page        */
/*      boundaries.                      */
/*                                       */
/* Return Values:                        */
/*   G_OK       success                  */
/*   G_CLIP     line does not fall in    */
/*                page at all            */
/*****************************************/
static int Gclip(x1,y1,x2,y2)
int *x1,*y1,*x2,*y2;
{
        unsigned code1, code2;  /* codes for endpoints */

        while (1) {
           code1 = Gencodepoint(*x1,*y1);
           code2 = Gencodepoint(*x2,*y2);
           if (!code1 && !code2)
              return (G_OK);
           if (code1 & code2)
              return (G_CLIP);
           Gcsclip(x1,y1,x2,y2,code1,code2);
           }
}

/*****************************************/
/* Gcsclip()                             */
/*      Part of Cohen-Sutherland         */
/*      Clipping algorithm.              */
/*                                       */
/* Return Values:                        */
/*   none                                */
/*****************************************/
static Gcsclip(x1,y1,x2,y2,code1,code2)
int *x1,*y1,*x2,*y2;
unsigned code1,code2;
{
        unsigned code;
        int *thisx, *thisy, *otherx, *othery;
        float slope;

        if (*y2 == *y1) {
           if (*x1 < minx) *x1 = minx;
           if (*x2 < minx) *x2 = minx;
           if (*x1 > maxx) *x1 = maxx;
           if (*x2 > maxx) *x2 = maxx;
           return;
           }
```

```
        if (*x2 == *x1) {
            if (*y1 < miny) *y1 = miny;
            if (*y2 < miny) *y2 = miny;
            if (*y1 > maxy) *y1 = maxy;
            if (*y2 > maxy) *y2 = maxy;
            return;
            }

        slope = (float) (*y2 - *y1) / (*x2 - *x1);

        if (code1 == 0) {
            code = code2;
            thisx = x2; thisy = y2;
            otherx = x1; othery = y1;
            }
        else {
            code = code1;
            thisx = x1; thisy = y1;
            otherx = x2; othery = y2;
            }

        if (code & 0x08) {
            *thisx = (maxy - *othery)/slope + *otherx;
            *thisy = maxy;
            }
        else if (code & 0x04) {
            *thisx = (miny - *othery)/slope + *otherx;
            *thisy = miny;
            }
        else if (code & 0x02) {
            *thisx = maxx;
            *thisy = slope*(maxx - *otherx) + *othery;
            }
        else {
            *thisx = minx;
            *thisy = slope*(minx - *otherx) + *othery;
            }
}

/*****************************************/
/* Gline()                               */
/*       Draws a line between (x1,y1)    */
/*         and (x2,y2). Will perform     */
/*         clipping if needed.           */
/*                                       */
/* Return Values:                        */
/*    G_OK        success                */
/*    G_NOINIT    not initialized w/ open */
/*    G_CLIP      line was completely    */
/*                  clipped out          */
/*****************************************/
Gline(x1,y1,x2,y2,linestyle)
int x1,y1,x2,y2;
```

```
unsigned linestyle;
{
        int rval;                       /* saves ret value of Gclip */
        int dx,dy;                      /* differences of x,y */
        int ix,iy;                      /* absolute values of dx,dy */
        int inc;                        /* greater of dx,dy */
        int plotx, ploty;               /* current point being plotted */
        int plotflag;                   /* boolean flag */
        int i;                          /* index variable */
        int x,y;                        /* used in computations */

        /* check to see if a Gopen has been done */
        if (map == HUGENULL)
           return (G_NOINIT);

        /* check endpoints for bounds */
        if ((x1<minx) | (x1>maxx) | (y1<miny) | (y1>maxy) |
                (x2<minx) | (x2>maxx) | (y2<miny) | (y2>maxy))
           if (rval = Gclip(&x1,&y1,&x2,&y2))
              return (rval);

        /* Bresenham's Line-Drawing Algorithm */
        /* adapted from pseudocode presented in
           "Graphics in C" by Nelson, Johnson */
        dx = x2-x1;
        dy = y2-y1;
        ix = abs(dx);
        iy = abs(dy);
        inc = max(ix,iy);

        plotx = x1; ploty = y1;
        x = 0; y = 0;

        /* if first bit of linestyle is set, then set pixel */
        /* the rest of the manipulation performs a circular shift */
        if (linestyle & 0x0001) {
           Gsetpixel(plotx,ploty);
           linestyle = (linestyle>>1) | 0x8000;
           }
        else
           linestyle >>= 1;


        for (i=0; i<= inc; i++) {
           x += ix;
           y += iy;

           plotflag = 0;

           if (x > inc) {
              plotflag++;
              x -= inc;
              if (dx>0) plotx++;
              if (dx<0) plotx--;
```

```
            }

        if (y > inc) {
            plotflag++;
            y -= inc;
            if (dy>0) ploty++;
            if (dy<0) ploty--;
            }

        if (plotflag)
            if (linestyle & 0x0001) {
                Gsetpixel(plotx,ploty);
                linestyle = (linestyle>>1) | 0x8000;
                }
            else
                linestyle>>=1;
        }

    return (G_OK);
}
```

```
/*************************************************************/
/*                                                           */
/*       DOT MATRIX PRINTER GRAPHICS LIBRARY                 */
/*                                                           */
/*                                                           */
/*       Author: Christopher Bock                            */
/*                                                           */
/*       Course: Honors College Thesis, ID 499               */
/*                                                           */
/*               Ball State University                       */
/*                                                           */
/*               Spring Semester 1989                        */
/*                                                           */
/*************************************************************/

/*************************************************************/
/*               RASTER VERSION                              */
/*************************************************************/

/* needed for farmalloc(), farfree() */
#include <alloc.h>
/* needed for biosprint() */
#include <bios.h>

#define max(x,y)        ((x)>(y)?(x):(y))
#define abs(x)          ((x)>0?(x):-(x))
/*****************************************/
/*      ERROR CODES                      */
/*****************************************/
#define G_OK            0
#define G_NOINIT        1
#define G_NOMEM         2
#define G_NOTRDY        3
#define G_BOUNDS        4
#define G_CLIP          5

/*****************************************/
/*      CONSTANTS                        */
/*****************************************/
#define TRUE            1
#define FALSE           0
#define ESC             27

/* for printer densities */
#define LORES           60
#define HIRES           120

/* various printer constants */
#define LPI             9
#define DPL             8
#define VERTDPI         72

/* define labels for printers */
#define PRN             0
```

```
#define LPT1            0
#define LPT2            1
#define LPT3            2

/* line styles */
#define G_SOLID         0xFFFF
#define G_DASHED        0xF0F0
#define G_DOTTED        0xCCCC

/*****************************************/
/*      GLOBAL VARIABLES                 */
/* all global variables are static so    */
/* they are unknown/unaccessible         */
/* outside of this file                  */
/*****************************************/

static int density;                     /* printer density */
static char * linemap = NULL;           /* pointer to bitmap of line */
static int minx, maxx;                  /* minimum/maximum for x */
static int miny, maxy;                  /* minimum/maximum for y */
static int pageheight;                  /* height of bitmap in lines */
static int pagewidth;                   /* width of bitmap in pixels */

static struct LINESTRUCT {
        int x1,y1,x2,y2;                /* coordinates of endpoints */
        unsigned linestyle;             /* line style */
        struct LINESTRUCT far *next;    /* prtr to next node */
        } far *headptr;

#define FARNULL ((struct LINESTRUCT far *) 0)


/*****************************************/
/* Gopen()                               */
/*      sets up a bitmap of the size     */
/*      needed for the 'density'         */
/*      provided.                        */
/*                                       */
/* Return Values:                        */
/*   G_OK       success                  */
/*   G_NOMEM    not enough memory        */
/*****************************************/
int Gopen(dens,width,height)
int dens;
float width,height;
{
        /* save density in global variable */
        density = dens;

        /* calculate page height & width in suitable units */
        pageheight = (int) height * LPI;        /* lines */
        pagewidth = (int) width * density;      /* pixels */

        /* allocate the memory needed for the line bitmap */
```

```
        if ((linemap = malloc(pagewidth)) == NULL)
           return (G_NOMEM);

        /* calculate minimums & maximums for x,y */
        minx = miny = 0;
        maxx = pagewidth-1;
        maxy = (pageheight * DPL)-1;

        /* clear linked list of lines */
        headptr = FARNULL;
        Gclear();

        return (G_OK);
}

/*****************************************/
/* Gclose()                              */
/*      releases the memory allocated    */
/*      to the bitmap.                   */
/*                                       */
/* Return Values:                        */
/*   G_OK        success                 */
/*   G_NOINIT    not initialized w/ open */
/*****************************************/
int Gclose()
{
        /* check to see if a Gopen has been done */
        if (linemap == NULL)
           return (G_NOINIT);

        /* clear out line list */
        Gclear();

        /* release memory from bitmap */
        free(linemap);

        /* mark linemap as pointing to NULL */
        linemap = NULL;

        return (G_OK);
}


/*****************************************/
/* Gclear()                              */
/*      clears the linked list to an     */
/*      'empty' condition. (no lines     */
/*      stored).                         */
/*                                       */
/* Return Values:                        */
/*   G_OK        success                 */
/*   G_NOINIT    not initialized w/ open */
/*****************************************/
int Gclear()
```

```
{
        long i;                     /* index variable */
        struct LINESTRUCT far *temp1, far *temp2;  /* temp pointers */

        /* check to see if a Gopen has been done */
        if (linemap == NULL)
           return (G_NOINIT);

        /* traverse linked list, freeing each node */
        temp1 = headptr;
        while (temp1 != FARNULL) {
           temp2 = temp1->next;
           farfree(temp1);
           temp1 = temp2;
           }

        /* set pointer to head of linked list to NULL */
        headptr = FARNULL;

        /* return an 'OK' condition */
        return (G_OK);
}


/*****************************************/
/* Gprint()                              */
/*      prints the page to the printer.  */
/*                                       */
/* Return Values:                        */
/*   G_OK        success                 */
/*   G_NOINIT    not initialized w/ open */
/*   G_NOTRDY    prtr not ready          */
/*   G_BOUNDS    prtr num out of bounds  */
/*****************************************/
Gprint(prtr)
int prtr;       /* printer number (0=LPT1,1=LPT2,2=LPT3) */
{
        int lines, columns;     /* index variables */
        char *tmpmap;           /* temp prtr to linemap */
        char prcode;            /* control code for graphics */
        int x1,y1,x2,y2;        /* endpoints for lines */
        int topy, boty;         /* y of top & bottom pixel on line */
        long longres;           /* used to calculate long result */
        struct LINESTRUCT far *temp;    /* pointer to line node */

        /* check to see if a Gopen has been done */
        if (linemap == NULL)
           return (G_NOINIT);

        /* check to see if printer number is valid */
        if ((prtr < 0) || (prtr > 2))
           return (G_BOUNDS);

        /* see if printer is ready */
```

```
if (Gprinterstatus(prtr))
   return (G_NOTRDY);

/* initialize the printer */
Gprinterinit(prtr);

/* set the printer up for correct vertical spacing */
Gprintchar(ESC,prtr); Gprintchar('A',prtr); Gprintchar(8,prtr);

/* go through each line of the page */
for (lines=0; lines<pageheight; lines++) {

   /* zero out line */
   Gzeroline();

   /* calculate y values for top & bottom pixels */
   topy = lines * DPL;
   boty = topy + DPL - 1;

   /* traverse linked list of lines */
   temp = headptr;
   while (temp != FARNULL) {

      /* see if current line falls within printline */
      if ((temp->y1 <= boty) && (temp->y2 >= topy)) {

         /* check for a horiz line (division by 0 error) */
         if (temp->y1 == temp->y2) {
            x1 = temp->x1; x2 = temp->x2;

            if (temp->y1 < topy)
               y1 = topy;
            else
               y1 = temp->y1;

            if (temp->y2 > boty)
               y2 = boty;
            else
               y2 = temp->y2;
            }

         else { /* non-horizontal */

            if (temp->y1 < topy) {
               longres = ((long)(topy - temp->y1)*
                         (long)(temp->x2 - temp->x1)/
                         (long)(temp->y2 - temp->y1))
                         + temp->x1;
               x1 = (int) longres;
               y1 = topy;
               }
            else {
               x1 = temp->x1;
               y1 = temp->y1;
```

```
            }

         if (temp->y2 > boty) {
            longres = ((long)(boty+1 - temp->y1)*
                      (long)(temp->x2 - temp->x1)/
                      (long)(temp->y2 - temp->y1))
                      + temp->x1;
            x2 = (int) longres;
            y2 = boty+1;
            }
         else {
            x2 = temp->x2;
            y2 = temp->y2;
            }

         }

      if (bresenham(x1,y1-topy,x2,y2-topy,&temp->linestyle))
         printf("Bresenham error: (%d,%d)(%d,%d) (%d,%d)(%d,%d)\n",
             x1,y1-topy,x2,y2-topy,
             temp->x1,temp->y1,temp->x2,temp->y2);
      }

   temp = temp->next;

   }

   /* check for a blank line */
   if (!Gblankline(linemap)) {

      /* send the graphics initialization string */
      switch (density) {
         case LORES: prcode = 'K'; break;
         case HIRES: prcode = 'L'; break;
         }

      Gprintchar(ESC,prtr); Gprintchar(prcode,prtr);

      /* send low order, then high order byte */
      Gprintchar(pagewidth & 0x00FF,prtr);
      Gprintchar(pagewidth >> 8,prtr);

      /* do the entire row */
      tmpmap = linemap;
      for (columns=0; columns<pagewidth; columns++)
         Gprintchar(*tmpmap++,prtr);
      }

   /* advance to the next line */
   Gprintchar(13,prtr); Gprintchar(10,prtr);
}

Gprintchar(12,prtr); /* go to TOF */
Gprintchar(ESC,prtr);        /* send software reset to printer */
```

```
        Gprintchar('@',prtr);
}

/*****************************************/
/* Gblankline()                          */
/*      checks to see if the line is     */
/*      blank (all zero's)               */
/*                                       */
/* Return Values:                        */
/*   TRUE       line is blank            */
/*   FALSE      line is not blank        */
/*****************************************/
static int Gblankline(bufprtr)
char far * bufprtr;
{
        int i;          /* index variable */

        i = pagewidth;
        while (i--)
          if (*bufprtr++)
             return (FALSE);

        return (TRUE);
}

/*****************************************/
/* Gzeroline()                           */
/*      blanks out the line (sets all    */
/*      bytes to zero)                   */
/*                                       */
/* Return Values:                        */
/*   G_OK       always                   */
/*****************************************/


static int Gzeroline()
{
        int i;
        char *lineprtr;

        lineprtr = linemap;
        for (i=0; i<pagewidth; i++)
           *lineprtr++ = '\0';

        return(G_OK);
}

/*****************************************/
/* Gprinterstatus()                      */
/*      returns status of the printer.   */
/*                                       */
/* Return Values:                        */
/*   0          printer ready            */
/*   non-0      problem at prtr          */
/*****************************************/
static int Gprinterstatus(prtnum)
```

```
int prtnum;
{
        return(biosprint(2,0,prtnum) & 0x29);
}


/*****************************************/
/* Gprinterinit()                        */
/*      Uses BIOS to initialize printer  */
/*                                       */
/* Return Values:                        */
/*   G_OK       success                  */
/*****************************************/


static int Gprinterinit(prtnum)
int prtnum;
{
        biosprint(1,0,prtnum);
        return (G_OK);
}


/*****************************************/
/* Gprintchar()                          */
/*      prints a byte to the printer     */
/*                                       */
/* Return Values:                        */
/*   G_OK       success                  */
/*   G_NOTRDY   prtr not ready           */
/*   G_ESC      user pressed ESC         */
/*****************************************/
static int Gprintchar(ch,prtnum)
char ch;
int prtnum;
{
        biosprint(0,ch,prtnum);
        return (G_OK);
}

/*****************************************/
/* Gencodepoint()                        */
/*      Part of Cohen-Sutherland         */
/*      Clipping algorithm.              */
/*                                       */
/* Return Values:                        */
/*   encoded value for endpoint          */
/*****************************************/
static unsigned Gencodepoint(x,y)
int x,y;
{
        unsigned code = 0;      /* used for constructing code */

        if (x<minx) code |= 0x01;
        if (x>maxx) code |= 0x02;
        if (y<miny) code |= 0x04;
```

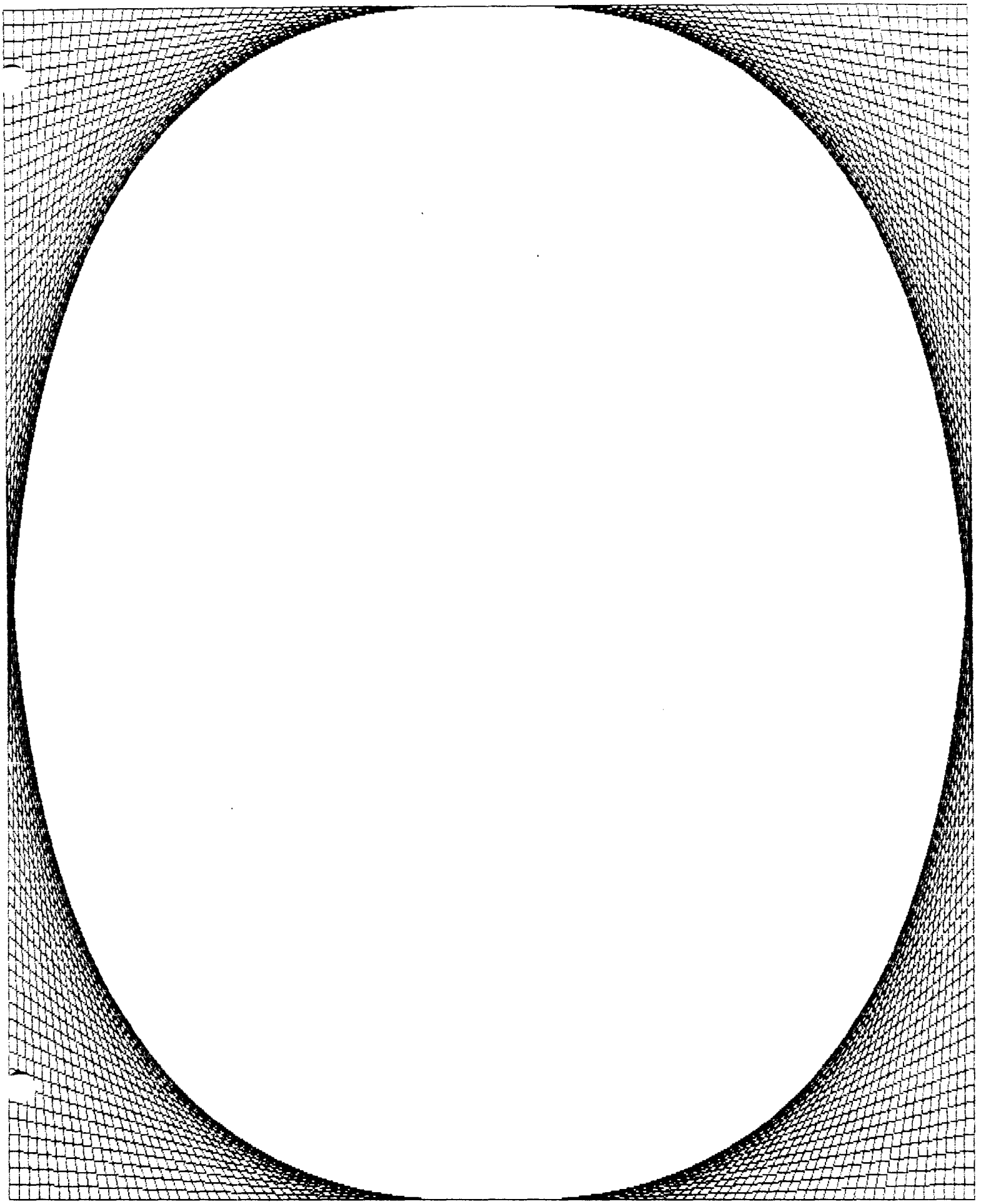
```
        if (y>maxy) code != 0x08;

        return (code);
}

/*****************************************/
/* Gclip()                               */
/*      Cohen-Sutherland algorithm       */
/*      Clips endpoints of a line that   */
/*      falls outside of the page        */
/*      boundaries.                      */
/*                                       */
/* Return Values:                        */
/*   G_OK       success                  */
/*   G_CLIP     line does not fall in    */
/*                page at all            */
/*****************************************/
static int Gclip(x1,y1,x2,y2)
int *x1,*y1,*x2,*y2;
{
        unsigned code1, code2;  /* codes for endpoints */

        while (1) {
           code1 = Gencodepoint(*x1,*y1);
           code2 = Gencodepoint(*x2,*y2);
           if (!code1 && !code2)
              return (G_OK);
           if (code1 & code2)
              return (G_CLIP);
           Gcsclip(x1,y1,x2,y2,code1,code2);
           }
}


/*****************************************/
/* Gcsclip()                             */
/*      Part of Cohen-Sutherland         */
/*      Clipping algorithm.              */
/*                                       */
/* Return Values:                        */
/*   none                                */
/*****************************************/
static Gcsclip(x1,y1,x2,y2,code1,code2)
int *x1,*y1,*x2,*y2;
unsigned code1,code2;
{
        unsigned code;
        int *thisx, *thisy, *otherx, *othery;
        float slope;

        if (*y2 == *y1) {
           if (*x1 < minx) *x1 = minx;
           if (*x2 < minx) *x2 = minx;
           if (*x1 > maxx) *x1 = maxx;
```

```
            if (*x2 > maxx)-*x2 = maxx;
            return;
            }

        if (*x2 == *x1) {
            if (*y1 < miny) *y1 = miny;
            if (*y2 < miny) *y2 = miny;
            if (*y1 > maxy) *y1 = maxy;
            if (*y2 > maxy) *y2 = maxy;
            return;
            }

        slope = (float) (*y2 - *y1) / (*x2 - *x1);

        if (code1 == 0) {
            code = code2;
            thisx = x2; thisy = y2;
            otherx = x1; othery = y1;
            }
        else {
            code = code1;
            thisx = x1; thisy = y1;
            otherx = x2; othery = y2;
            }

        if (code & 0x08) {
            *thisx = (maxy - *othery)/slope + *otherx;
            *thisy = maxy;
            }
        else if (code & 0x04) {
            *thisx = (miny - *othery)/slope + *otherx;
            *thisy = miny;
            }
        else if (code & 0x02) {
            *thisx = maxx;
            *thisy = slope*(maxx - *otherx) + *othery;
            }
        else {
            *thisx = minx;
            *thisy = slope*(minx - *otherx) + *othery;
            }
}


/*****************************************/
/* Gline()                               */
/*       Draws a line between (x1,y1)    */
/*         and (x2,y2). Will perform     */
/*         clipping if needed.           */
/*                                       */
/* Return Values:                        */
/*    G_OK        success                */
/*    G_NOINIT    not initialized w/ open */
/*    G_CLIP      line was completely    */
```

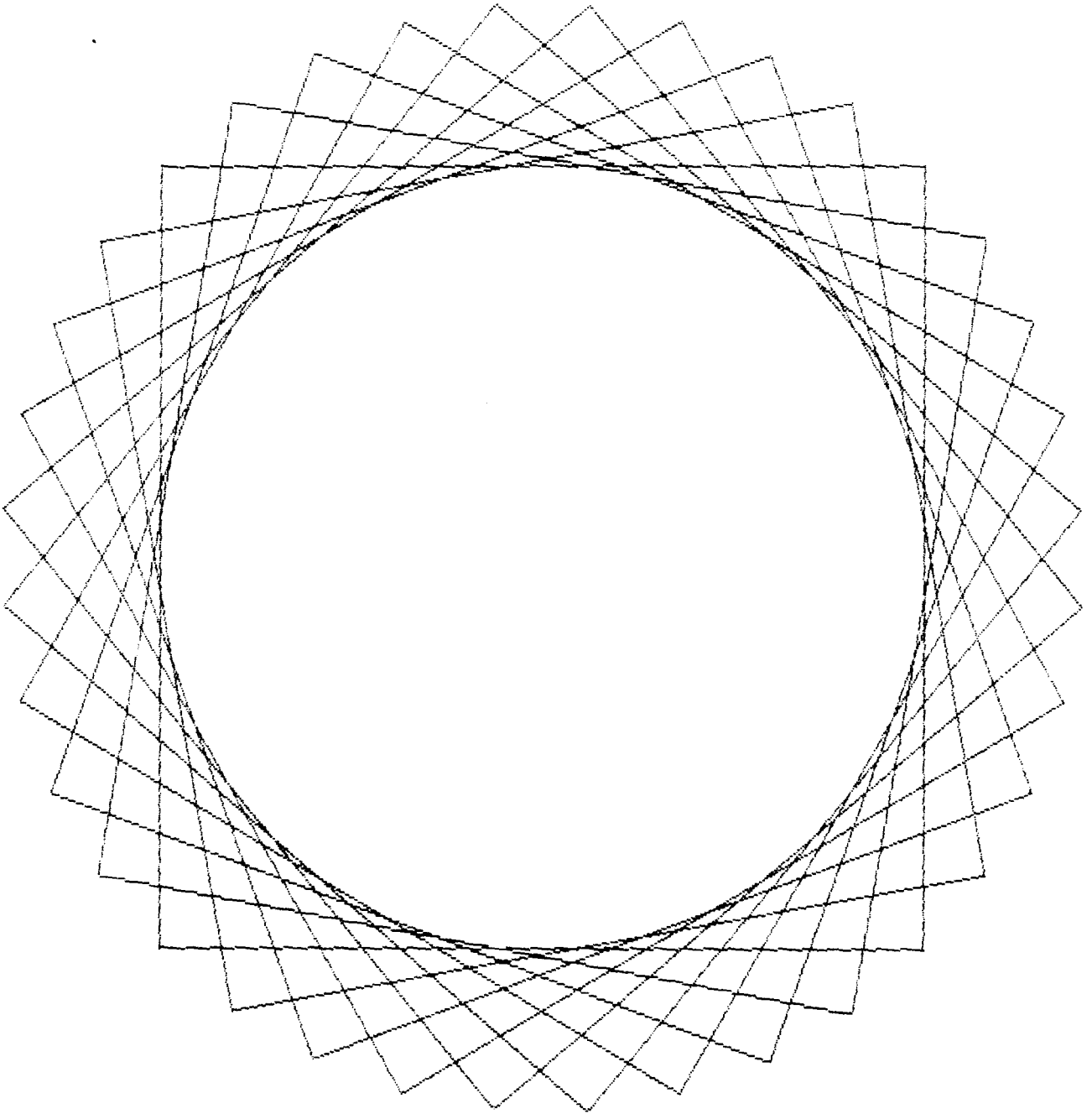
```
/*                  clipped out                 */
/*******************************************/
Gline(x1,y1,x2,y2,linestyle)
int x1,y1,x2,y2;
unsigned linestyle;
{
        int rval;                   /* saves ret value of Gclip */

        struct LINESTRUCT far *temp;


        /* check to see if a Gopen has been done */
        if (linemap == NULL)
           return (G_NOINIT);

        /* check endpoints for bounds */
        if ((x1<minx) | (x1>maxx) | (y1<miny) | (y1>maxy) |
              (x2<minx) | (x2>maxx) | (y2<miny) | (y2>maxy))
           if (rval = Gclip(&x1,&y1,&x2,&y2))
              return (rval);

        /* create a node */
        if ((temp = farmalloc(sizeof(struct LINESTRUCT))) == FARNULL)
           return (G_NOMEM);

        /* insert values, sorting endpoints at the same time */
        if (y1 < y2) {
           temp->x1 = x1; temp->y1 = y1;
           temp->x2 = x2; temp->y2 = y2;
           }
        else {
           temp->x1 = x2; temp->y1 = y2;
           temp->x2 = x1; temp->y2 = y1;
           }

        /* insert linestyle */
        temp->linestyle = linestyle;

        /* link this node into the list at the beginning */
        temp->next = headptr;
        headptr = temp;

        /* all is well... return OK status */
        return (G_OK);
}

/*******************************************/
/* Gsetpixel()                             */
/*       sets the pixel designated by      */
/*         parameters x,y                  */
/*                                         */
/* Return Values:                          */
/*    G_OK        success                  */
/*    G_BOUNDS    pixel is out of bounds   */
```

```
/*****************************************/
static Gsetpixel(x,y)
int x,y;
{
        /* error checking */
        if ((y<0) || (y>7) || (x<0) || (x>maxx))
           return(G_BOUNDS);

        /* set appropriate pixel */
        *(linemap+x) |= (0x80 >> y);

        /* return OK status */
        return (G_OK);

}

/*****************************************/
/* bresenham()                           */
/*      calculates the points needed to  */
/*      plot the line (x1,y1)-(x2,y2)    */
/*      using bresenham's algorithm.     */
/*                                       */
/* Return Values:                        */
/*   G_OK       success                  */
/*****************************************/
static bresenham(x1,y1,x2,y2,linestyle)
int x1,y1,x2,y2;
unsigned *linestyle;
{
        int dx,dy;              /* differences of x,y */
        int ix,iy;              /* absolute values of dx,dy */
        int inc;                /* greater of dx,dy */
        int plotx, ploty;       /* current point being plotted */
        int plotflag;           /* boolean flag */
        int i;                  /* index variable */
        int x,y;                /* used in computations */

        /* Bresenham's Line-Drawing Algorithm */
        /* adapted from pseudocode presented in
           "Graphics in C" by Nelson, Johnson */
        dx = x2-x1;
        dy = y2-y1;
        ix = abs(dx);
        iy = abs(dy);
        inc = max(ix,iy);

        plotx = x1; ploty = y1;
        x = 0; y = 0;

        if (*linestyle & 0x0001) {
           Gsetpixel(plotx,ploty);
           *linestyle = (*linestyle>>1) | 0x8000;
           }
        else
```

```
        *linestyle>>=1;

    for (i=0; i<= inc; i++) {
        x += ix;
        y += iy;

        plotflag = 0;

        if (x > inc) {
            plotflag++;
            x -= inc;
            if (dx>0) plotx++;
            if (dx<0) plotx--;
            }

        if (y > inc) {
            plotflag++;
            y -= inc;
            if (dy>0) ploty++;
            if (dy<0) ploty--;
            }

        if (plotflag)
            if (*linestyle & 0x01) {
                Gsetpixel(plotx,ploty);
                *linestyle = (*linestyle>>1) | 0x8000;
                }
            else
                *linestyle>>=1;
        }

    return (G_OK);
}
```



```
/*****************************************/
/*       SOURCE CODE FOR BORDER          */
/*          FORMED FROM LINES            */
/*                                       */
/* Chris Bock, ID 499 Honors Thesis      */
/*****************************************/

main()
{
        int i;
        int botval;

        if (Gopen(HIRES,8.0,10.0))
           printf("Unable to open for graphics\n");

        Gline(minx,miny,maxx,miny,G_SOLID);
        Gline(maxx,miny,maxx,maxy,G_SOLID);
        Gline(maxx,maxy,minx,maxy,G_SOLID);
        Gline(minx,maxy,minx,miny,G_SOLID);

        botval = (pageheight * DPL) - 1;

        for (i=0; i<450; i+=10) {
           Gline(i,0,0,450-i,G_SOLID);
           Gline(pagewidth-i-1,0,pagewidth-1,450-i,G_SOLID);
           Gline(i,botval,0,botval-450+i,G_SOLID);
           Gline(pagewidth-i-1,botval,pagewidth-1,botval-450+i,G_SOLID);
           }

        if (Gprint(LPT1))
           printf("Error occured during printout\n");

        exit(0);
}
```



```
/*****************************************/
/*       SOURCE CODE FOR CIRCLE          */
/*         FORMED FROM SQUARES           */
/*                                       */
/* Chris Bock, ID 499 Honors Thesis      */
/*****************************************/

#include "graphlib.h"
#include <math.h>

#define CVT (3.14159/180.0)
#define MIDX 450
#define MIDY 350
#define XFACT 400.0
#define YFACT 250.0

main()
{
        double i;
        int x1,y1,x2,y2;
        int botval;

        if (Gopen(HIRES,8.0,10.0))
           printf("Unable to open for graphics\n");

        for (i=0.0; i<90.0; i+=10.0) {
           x1 = (int) (XFACT * cos(i*CVT));
           y1 = (int) (YFACT * sin(i*CVT));
           x2 = (int) (XFACT * cos((i+90)*CVT));
           y2 = (int) (YFACT * sin((i+90)*CVT));

           Gline(MIDX+x1,MIDY+y1,MIDX+x2,MIDY+y2,G_DOTTED);
           Gline(MIDX+x2,MIDY+y2,MIDX-x1,MIDY-y1,G_DOTTED);
           Gline(MIDX-x1,MIDY-y1,MIDX-x2,MIDY-y2,G_DOTTED);
           Gline(MIDX-x2,MIDY-y2,MIDX+x1,MIDY+y1,G_DOTTED);
           }

        if (Gprint(LPT1))
           printf("Error occured during printout\n");

        exit(0);
}
```